Brown Bag

Introduction to Akka

by Hendri Karisma
Sr. Software Development at blibli.com

Reactive Programming

Before, managed servers and container -> (now) reactive applications :

- React to events (event-driven)
- React to load (scalable)
- React to failures (resilient)
- React to users (responsive)

Actor Mode

The actor model in computer science is a mathematical model of concurrent computation that treats "actors" as the universal primitives of concurrent computation: in response to a message that it receives, an actor can make local decisions, create more actors, send more messages, and determine how to respond to the next message received. The actor model originated in 1973. (wikipedia)

Actor Model

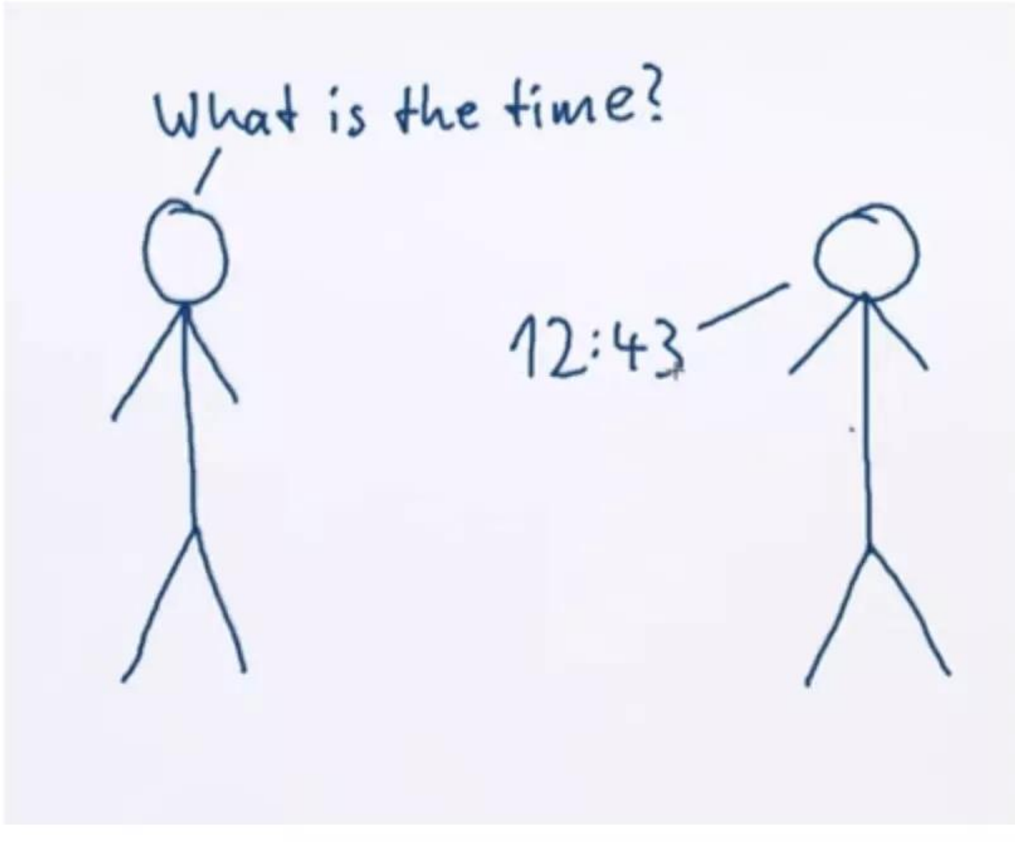
- Carl Hewitt et al, 1973: Actor invited for research on artificial intelligence
- Gul Agha, 1986: Actor languages and communication patterns
- Ericsson, 1995: first commercial use in Erlang/OTP for telecommunication platform (5.2 minutes of downtime/year)
- Philipp Haller, 2006: implementation in scala standard librai
- Jonas Boner, 2009: creation of Akka

What is Akka

Akka is a toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications on the JVM. (http://akka.io/)

What is Actor

The actor model represents objects and their interactions, resembling human organisation and built upon the law of physics



Why Actor

When it was all about the *megahertz* and nobody was even thinking about the day when it would be all about the *number of cores*.

hardware with a ton of cores and huge potential for concurrency, what tools do you have to go with it? Threads, locks, mutexes, critical sections, synchronized methods, and all of their brothers, sisters, cousins, and pets….

You and I deal with concurrency every single day. Let's face it, your entire life is a series of interrupts and a collection of requests and responses.

Actor Summary

- Optimizing the use of threads requires non-blocking APIs, which are rare in the wild and are also historically cumbersome to write.

- Dealing with errors in an asynchronous system can be difficult to manage. Traditional error-handling strategies (exceptions and return codes) do not translate well when inside a concurrent application.

- Controlling access to your data requires constant vigilance. Ensuring that race conditions and deadlocks are eliminated can verge on the impossible.

- Setting up a signaling mechanism between two objects is required in order to eliminate race conditions and speed up wait times.

- The complexity of most common concurrent applications can increase the cost of refactoring to a fairly high degree. This makes complex concurrent applications accrue technical debt at a much faster rate than their sequential counterparts.

Parallelism vs Concurrency

Parallelism The act of modifying a seemingly sequential algorithm into a set of mutually independent parts that can be run simultaneously, either on multiple cores or multiple machines. For example, you can multiply thousands of matrices sequentially but you can also do this in parallel if you break them up into a hierarchy of multiplications.

Concurrency This is all that other stuff, also known as *life*. It's the act of an application, which has many dependent or independent algorithms, running through multiple threads of execution simultaneously. The easiest example is that of a web service, such as Twitter. Twitter is highly event driven, taking in tweets from millions of concurrent users as well as events from its own internal systems. All of this stuff happens concurrently.

Akka has solid solutions for both parallelism and concurrency.

Concurrency

Shared-state concurrency

- Create some object that carries some data.
- Create some functions to operate on that data.
- Realize it's slow. Create threads (or an ExecutorService) and refactor your code so it can run concurrently.
- Find out that you have a ton of race conditions. Protect the data with synchronization primitives.
- Scratch your head about why only 10% of your cores are being utilized. Eventually you blame the synchronization.
- Refactor again to try and increase concurrency. And so on. . .

* Deadlocks, race conditions, memory corruption, scalability bottlenecks, resource contention, and *Heisenbug*

Message Passing

- Has the benefit of being easier to reason about, and also easier to implement in a distributed computing system.
- Threads communicate and synchronize using three key abstractions: channels, mailboxes, and events.
- The prominent mathematical models of message passing are the Actor model and Pi calculus.
- Concurrency refers to the idea of having many actors running independently, but not necessarily all at the same time.

Thanks…

sample code :

- bank account transfer : https://github.com/situkangsayur/bank-account

- link checker : https://github.com/situkangsayur/link-checker